INTERNATIONAL
APPLE CORE

P.O.BOX 976 DALY CITY, CALIFORNIA 94017

ATTACH-BIOS

FOR APPLE II PASCAL 1.1

BY BARRY HAYNES

"APPLE" is the registered trademark of Apple Computer, Inc. The **INTERNATIONAL APPLE CORE** is licensed by Apple Computer, Inc. to use certain of the latter's trademarks.

NOTE: EVERY EFFORT HAS BEEN MADE TO PROVIDE YOU ERROR-FREE PROGRAMS. However, you are reminded that neither the author nor the **International Apple Core** can accept any responsibility for any loss or damage, tangible or intangible, resulting from program errors or improper or inintended use of this information or programs.

ATTACH-BIOS

FOR APPLE II PASCAL 1.1

BY BARRY HAYNES

ATTACH-BIOS document for Apple II Pascal 1.1

By Barry Haynes

Jan 12, 1980

This document is intended for Apple II Pascal internal applications writers, Vendors and Users who need to attach their own drivers to the system or who need more detailed information about the 1.1 BIOS. It is divided into two sections, one explaining how to use the ATTACH utility available through technical support and the other giving general information about the BIOS. It is a good idea to read this whole document before assuming something is missing or hasn't been completely explained. This document is intended for more advanced users who already know a fair amount about I/O devices and how to write device drivers. It is not intended to be a simple step by step description of how to write your first device driver, nor does it claim to be a complete description of all there is to know about the Pascal BIOS.

The Apple Pascal UCSD system has various levels of I/O that are each responsible for different types of actions. It was divided at UCSD into these levels to make it easy to bring up the system on various processors and also various configurations of the same processor and yet have things look the same to the Pascal level regardless of what was below that level. The levels are:

LEVEL	TYPES OF IO ACTIONS
Pascal	READ & WRITE
	BLOCKREAD & BLOCKWRITE
	UNITREAD & UNITWRITE
	UNITCLEAR
	UNITSTATUS
RSP (Runtime Support Package)	This is part of the interpreter and is the middle man between the above types of I/O and the below types of I/O. All the above types are translated by the compiler and operating system into UNITREAD, UNITWRITE, UNITCLEAR and UNITSTATUS if they are not already in that form in the Pascal program. The RSP checks the legality of the parameters passed and reformats these calls into calls to the BIOS routines below. The RSP also expands DLE (blank suppression) characters, adds line feeds to carriage returns, checks for end of

file (CTRL C from CONSOLE:), monitors UNITRW control word commands, makes calls to attached devices if present, echoes to the CONSOLE:.

BIOS (Basic I/O Subsystem) This is the lowest level device driver routines. This is the level at which you can attach new drivers to replace or work with the regular system drivers and also attach drivers for devices that will be completely defined by you.

I.RECONFIGURING THE BIOS TO ADD YOUR OWN DRIVERS USING THE ATTACH UTILITY.

INTRODUCTION

With the Apple Pascal 1.1 System (both regular and runtime 1.1), there is an automatic method for you to configuer your own drivers into the system. This method requires you to write the drivers following certain rules and to use the programs ATTACHUD.CODE and SYSTEM.ATTACH provided through Apple Technical Support. At boot time, the initialization part of SYSTEM.PASCAL looks for the program SYSTEM.ATTACH on the boot drive. If it finds SYSTEM.ATTACH, it Xecutes it before Xecuting SYSTEM.STARTUP. SYSTEM.ATTACH will use the files ATTACH.DATA and ATTACH.DRIVERS which must also be on the boot disk. ATTACH.DATA is a file the developer will make using the program ATTACHUD. It tells SYSTEM.ATTACH the needed information about the drivers it will be attaching. ATTACH.DRIVERS is a file containing all the drivers to be attached and is constructed by the developer using the standard LIBRARY program. The drivers are put on the Pascal Heap below the point that a regular program can access it. They do take away Stack-Heap (= to the size of the drivers attached) space from that available to Pascal code files but this should not be a problem unless the drivers are very large or the code files very hungry in their use of memory. Since these drivers are configured into the system after the operating system starts to run, this method will not work for configuring drivers for devices that the system must cold boot from. Some of supporting code in the RSP, boot and Bios may make the task of bringing up boot drivers easier though. The advantages to this kind of setup are:

1. Software Vendors can use the ATTACHUD program to put their own drivers into the system at boot time. This will be invisible to the user.

2. There can be no problems losing drivers due to improper heap management since the drivers are put on the heap by the operating system and before any user program can allocate heap space.

3. This method does not freeze parts of the system to special

memory locations since it enforces the clean methodology of using relocatable drivers.

USING ATTACHUD

ATTACHUD.CODE will ask you questions about the drivers you want to attach to the system. It makes a file called ATTACH.DATA which tells SYSTEM.ATTACH which drivers to attach to the system, what unit numbers to attach them to and other information. The options covered by ATTACHUD are:

1. A driver can be attached to one of the system devices, then all I/O to this device (PRINTER: for example) will go to this new driver. In the case of a new driver for a disk device the user will have to specify which of the 6 standard disk units will go to this new driver. This will allow replacement of standard drivers with custom ones without having to restrict the I/O interface to UNITREAD and UNITWRITE as is the case with option 2.

2. A driver can be attached to one of 16 userdevices. I/O to these will be done with UNITREAD and UNITWRITE to device numbers 128-143.

3. A method will be included to allow the attached driver to start on an N byte boundry. The driver writer will be responsible for aligning his code from that point.

4. More than one unit can be attached to the same driver. This way only one copy of the driver resides in memory and I/O to all the attached units goes to this one driver. It is up to the driver to decide which unit's I/O it is doing. How this is done is explained below.

5. The initialize routine for any attached driver can be called by SYSTEM.ATTACH after it has attached the driver and before any programs can be Xecuted.

6. In case any of your programs use the Hires pages, you can specify in ATTACHUD that drivers must not be put on the heap over these areas. Your drivers would have to be quite large before they could possibly overlap the Hires pages.

Follow through this example of a session with ATTACHUD where the options available are completely described. First Xecute ATTACHUD:

You will be given the prompt:

```
Apple Pascal Attachud [1.1]

Enter name of attach data file:
```

This is asking for what you want the output file from this session

with ATTACHUD to be called. You could call it ATTACH.DATA or some other name and then rename it to ATTACH.DATA when you put it on the boot disk with SYSTEM.ATTACH.

If you ever get a message of the form:

```
ERROR => some error
Try again (RETURN to exit program):
```

then just retype what was requested on the previous prompt after deciding what mistake you made while typing it the first time.

The next prompt is:

```
These next questions will determine if
attached drivers can reside in the hires
pages. It will be assumed they can for the
page in question if you answer no to the
prompt for that page.
Will you ever use the (2000.3FFF hex)
hires page?
```

Followed by:

```
Will you ever use the (4000.5FFF hex)
hires page?
```

You should answer yes to the question for a particular Hires page if you will ever be running a program that uses that Hires page while the drivers are Attached. You don't want the possibility of your driver residing in the Hires page if that page will be clobbered by one of your programs. After answering the Hires questions you will be asked the following questions once for each driver you will be attaching:

```
What is the name of this driver? This
must be the .PROC name in its assembly
source (RETURN to exit program):
```

This must be the name of one of the drivers in the ATTACH.DRIVERS that will be used with this ATTACH.DATA. The length of this name must not be more than 8 characters. After entering the name you will be asked:

```
Which unit numbers should refer to this
device driver?

Unit number (RETURN to abort program):
```

You must enter a unit number in the range 1,2,4..12,128..143 or will be given an error message. You cannot attach a character unit (CONSOLE:, PRINTER: or REMOTE:) to the same driver as a block structured unit and if you try you will be given the message:

```
You can''t attach a character unit and
a block unit to the same driver. I
will remove the last unit# you entered.
Type RETURN to continue:
```

If you don't get the above error, you will be asked:

```
Do you want this unit to be
initialized at boot time?
```

A yes response will put the unit number just entered on a list of units that SYSTEM.ATTACH will call UNITCLEAR on after attaching all the drivers. This gives you a way to have the system make an initialize call on your attached unit at boot time. A no response will mean that no boot time init call will be made on this unit to the driver you just attached.

You will be eventually asked:

```
Do you want another unit number to refer
to this device driver?:
```

A yes response will get you to the Unit number prompt again and a no response will get you to the prompt:

```
Do you want this driver to start on a
certain byte boundary?
```

A yes here will give you more prompts:

```
The boundry can be between 0 and 256.
  0=>Driver can start anywhere.(default)
  8=>Driver starts on 8 byte boundary.
  N=>Driver starts on N byte boundary.
256=>Driver starts on 256 byte PAGE boundary.
Enter boundary (RETURN to exit program):
```

And the last line of the prompt will repeat until you enter a boundary in the correct range. The boundary refers to the memory location where the first byte of the driver is loaded. If your driver needs to be aligned on some N byte boundary you can assure it will be using this mechanism. if you know how the driver's origin is aligned, You can align internal parts of your driver however you want. Finally you will get to the prompt:

```
Do you want to attach another driver?
```

And if you answer Yes to this you will return to the 'What is the name of this driver' prompt and answering No will end the program, saving the data file you have made.

THE DRIVER

Drivers must be written in assembly using the Pascal Assembler. They must not use the .ABSOLUTE option, so the drivers can be relocated as they are brought in by the system. Each driver must be assembled separately with no external references. When all drivers are assembled, use the LIBRARY program (in the same way you would use it to put units into a library) to put all the drivers in one file.

Name this file SYSTEM.DRIVERS. See further explanation of making SYSTEM.DRIVERS below.

Considerations for all drivers:

1. Study the examples below as certain information is only documented there.

2. Refer to the Apple II Pascal memory map below and you will see that parts of the interpreter and BIOS reside in the same address range and are bank-switched. The system automatically folds in the BIOS for drivers added using ATTACH. Most of these drivers will have to make calls to CONCK if they want type ahead to continue to work properly. CONCK is the BIOS routine that monitors the keyboard. See the example drivers below to be sure you are doing this correctly. You cannot call CONCK through the CONCK vector at BF0A (see BIOS part of this document) because this call would go through the same mechanism used to get to your driver and the return address to Pascal would be lost.

3. All attached drivers must be written with one common entry point for read, write, init and status. The driver will use the Xreg contents to decide which type of I/O call this is and jump to the appropriate place within it's code. The Xreg is decoded as follows:

```
0 -->read (no bits set)
1 -->write (bit 0 set)
2 -->init (bit 1 set) { The Pascal statement
     UNITCLEAR(UNITNUMBER); makes an init call for
     unit UNITNUMBER }
4 -->status (bit 2 set)
```

4. The drivers must also pop a return address off the stack, save it and later push it to do a RTS when the driver is finished. All other parameters must be removed from the stack by the driver. For all calls, the return address will be the top word on the stack.

5. SYSTEM.ATTACH will make a copy of the normal system jump vector (the vector after the fold) and put this on the heap. There will be a pointer to this vector at 0E2. Your drivers can use this vector to get to the normal system drivers for device numbers 1..12. See example below.

6. All drivers must pass back a completion code in the X register corresponding to the table on page 280 of the 1.1 "Apple II Apple Pascal Operating System Reference Manual".

7. In references below to parameters passed on the stack, all parameters are one word parameters so they require two bytes to be popped from the stack by the driver.

8. Control word format for Unitread & Unitwrite

```
bits 15..13 12..6 5 4 3 2 1..0
     user reserved type B type A nocrlf nospec reserved
     defined for future chars chars for future
     functions expansion expansion

     type B =0 ==>System will check for CTRL S & F from CONSOLE:
                  during the time of this Unitio call.
           =1 ==>System will not check for CTRL S & F during this
                  Unitio.
     type A =0 ==>If using Apple Keyboard, system will check for
                  CTRL A,Z,K,W & E from CONSOLE: during the period
                  of this Unitio.
           =1 ==>System will not check for the chars during
                  this Unitio.
     nocrlf =0 ==>line feeds are added to carriage returns by the
                  Interpreter.
           =1 ==>no line feeds are added ...
     nospec =0 ==>DLE's (blank suppression code) are expanded on
                  output and the EOF character is detected on input
           =1 ==>nothing special is done to DLE's on output and
                  EOF on input.

     default setting for all control word bits = 0.
```

9. Control word format for UNITSTATUS

```
bits 15..13 12..2 1 0
     user reserved for direction
     defined for future purpose

     direction =0 ==>Status of output channel is requested
               =1 ==>Status of input ...
     purpose =0 ==>Call is for unit status
             =1 ==>Call is for unit control
```

10. These are the new vectors and routines added to the BIOS to make attach work. The RSP, bootstrap, and readseg were also modified to allow for attaches.

```
UDJMPVEC ;Jump vector for user devices, offset=0 => unattached device.
         ;The correct addresses are initialized by SYSTEM.ATTACH
         ;See locations section of BIOS part below for pointers to
         ;this vector.
         JMP 0 ;Unit 128
         JMP 0 ;Unit 129
         .
         .
         .
         JMP 0 ;Unit 143

DISKNUM ;If high byte=FF then
         ; device is not a disk drive
         ;else
         ; if high byte=0 then
         ; device is a regular disk drive and low byte=drive #
```

```
            ; else
            ; driver for this disk drive has been attached by SYSTEM.ATTACH
            ; and the driver address is stored in this word.
            ; (Driver address has to be the address-1 for RTS in PSUBDR
            ; to work correctly, remember this for ATTACH. PSUBDR is
            ; listed below.)
            ;See locations section of BIOS part below for pointers to
            ;this vector.
            .WORD OFFFF ;Unit #1
            .WORD OFFFF ;Unit #2 (ATTACH would modify the words
            .WORD OFFFF ;Unit #3 for units 4,5,9..12 if a
            .WORD 0 ;Unit #4 different disk driver were
            .WORD 1 ;Unit #5 attached to any of them)
            .WORD OFFFF ;Unit #6
            .WORD OFFFF ;Unit #7
            .WORD OFFFF ;Unit #8
            .WORD 4 ;Unit #9
            .WORD 5 ;Unit #10
            .WORD 2 ;Unit #11
            .WORD 3 ;Unit #12

UDRWIS ;Routine to get to an attached driver through UDJMPVEC
            ;Assume unit# in Areg & operation to be performed in Xreg.
            ;See the jump vector in the BIOS sections to see how you
            ;get to this routine.
            STA TT1
            AND #7F ;Clear top bit of unit#
            STA TT2 ;Make address in UDJMPVEC table
            ASL A ;Address=Areg*3 + base of table
            CLC
            ADC TT2 ;Now we have (Areg*3).
            ADC #JVECTRS ;Add in low byte of base of table having
            STA TT2 ;no carry problem with only 16 UD's.
            LDA #0
            ADC JVECTRS+1 ;JVECTRS is a word pointing to the base
                                    ;of UDJMPVEC.
            STA TT2+1
            LDA TT1
            JMP @TT2


PSUBDR ;Routine to get to an attached driver through DISKNUM
            ;We assume on entry, Areg=unit#, Yreg=DISKNUM
            ;offset & Xreg=the command to be performed by the substituted
            ;disk driver.
            ;See the jump vector in the BIOS sections to see how you
            ;get to this routine.
            STA TT1 ;Save unit#.
            LDA DISKNUM-1,Y ;Store MSB of driver address.
            PHA
            LDA DISKNUM-2,Y ;Store LSB of driver address.
            PHA
            LDA TT1 ;Restore unit# to Areg.
            RTS ;Jump to substituted driver. This assumes
                                    ;the driver address in DISKNUM =
                                    ;(ADDRESS OF DRIVER)-1 for the RTS to work
```

Special considerations when attaching drivers for the system devices, unitnumbers 1..12.

A. Character Oriented Devices (Pass the character to be read-written in the A-register and make Bios calls one character at a time from RSP level. On entry, the unit number will be in the Y register in case you wanted to attach all character oriented devices to the same driver). If you attach REMOTE: & or PRINTER: to the same driver as CONSOLE:, all will have their jump vectors pointing to the start of the driver+3 bytes. See further discussion on this below.

Units 1 & 2 (CONSOLE: and SYSTERM:)

1. These must both go to the same driver.
2. The system CONCK routine will be patched to jump to the start of the driver. The CONCK routine gets characters entered at the keyboard and fills the type ahead buffer. See the example CONSOLE: driver below.
3. Because of item 2, the entry point for normal calls (not CONCK calls) to the attached driver will be 3 bytes beyond the start of the driver.
4. The interpreter takes care of expanding blank suppression codes (DLE's), echo to the screen, EOF (the end of file character), and adding line feeds to every carriage return. Your driver doesn't need to do this.
5. CONSOLE: read and write have only the return address on the stack. The stack for CONSOLE: init looks like:

```
POINTER TO BREAK VECTOR (This should be stored at
                         location BF16..BF17 by CONSOLE:
                         init.)
POINTER TO SYSCOM (This should be stored at
                   location F8..F9 by CONSOLE:
                   init.)
                  (Also at init time, the Flush
                   and Start/Stop conditions
                   should be set to normal and
                   the type-ahead queue should
                   be emptied.)
RETURN ADDRESS <--TOS (top of stack)
```

   The stack for CONSOLE: status looks like:

```
POINTER TO STATUS RECORD
CONTROL WORD
RETURN ADDRESS <--TOS
```

6. A status request should return, in the first word of the status record, the number of characters currently queued in the direction asked for. This is the number of characters in the type-ahead buffer. If no type-ahead is being used then output status should always return a 0 and input status a 1 if a char is waiting to be read, otherwise a 0.
7. Since we are using 7 bit ASCII codes, the CONSOLE: read routine should zero the high order bit of all characters it reads from the keyboard and passes back to Pascal (to the RSP). The CONSOLE: write routine should transfer all 8 bits as received from the RSP since many devices use 8 bit control codes.

8. The RSP will send both upper and lower case chars to the CONSOLE: write routine. The write routine should map the lower to upper if the device cannot handle lower case.
9. CONSOLE: Output Requirements:
   A. CR (OD hex) A carriage return should move the cursor to the beginning of the current line.
   B. LF (OA hex) A line feed should move the cursor to the next line but not change the column position. If the cursor is on the last line on the screen when a line feed is sent, the rest of the screen should scroll up one line and the bottom line be cleared.
   C. BELL (07 hex) A sound should be made if possible when the CONSOLE: gets 07. If making a sound is not possible then ignore the 07.
   D. SP (20 hex) Place a space at the current cursor position overwriting whatever is there. Move the cursor to the next column. If the cursor is on the last column of a line, it is best if the cursor stays where it is after the space fills that position. If the cursor is on the last column of the last line on the screen, it is also best if the cursor remains in that position and the screen does not scroll. These are the prefered actions of the cursor at end of line & end of screen; in the strict sense, the actions of the cursor in these circumstances are undefined.
   E. NUL (00 hex) When a Null is sent to the CONSOLE: from the RSP, the CONSOLE: should delay for the ammount of time required to write one character but the state of the screen should not change.
   F. All printable characters should be written to the screen and the cursor should move in the same way it does for SP.
   G. See the discussion on pages 199-215 in the 1.1 Operating System Reference Manual for further requirements and information.
10. CONSOLE: Input Requirements:
   A. The RSP takes care of echoing characters to the screen typed from the CONSOLE: keyboard.
      (below items optional The Start/Stop, Flush & Break chars are redefinable; see 9G above for more info.)
   B. The Start/Stop character is detected by CONCK and is used to stop all processing until the character is received a second time. When the character is received (see 9G above for more info) one should loop in CONCK continuing to process other characters until:
      1. the S/S char is received again
      2. the Break char is received

      In case 1, the suspended processing should continue as it was before the first S/S was typed. Action needed for the Break char is described below. The S/S char is never returned to the RSP and CONSOLE: type-ahead, if implemented, should continue during the suspended state. Offset from SYSCOM to this char is 85 decimal. (This and the next 2 chars are redefinable by the Setup program and SYSCOM is the system area that keeps track of this info. The pointer to the start of SYSCOM is passed to the CONSOLE: init routine and is stored at F8..F9 hex.)
   C. The Flush character will stop all output and echoing to the

CONSOLE: until it's second occurEnce (see 9G above). CONCK detects this and must set a flag to tell the CONSOLE: output routine to ignore characters while the flag is set. If the CONSOLE: is re-initialized or a Break char is received, the flush state should be turned off. Flush is never returned to the RSP. Flush only stops CONSOLE: output, other processing continues. Offset from SYSCOM to this char is 83 decimal.

D. The Break char should cause CONCK to jump to the location stored at BF16. This location is also passed to the CONSOLE: init routine which stores it at BF16. The break char is never returned to the RSP and it should remove the system from Stop or Flush mode if it is in either mode. Offset from SYSCOM to this char is 84 decimal.

E. Type-ahead should be implemented in CONCK by storing characters typed at the keyboard in a queue until they are requested by a CONSOLE: read from Pascal. When the queue fills, further characters should be ignored and a bell sounded when they are typed. The length of the queue should be at least 20 characters.

11. For more information on CONSOLE: requirements, see pages 199-216 of the 1.1 Operating System Reference Manual.

Unit 6 (the PRINTER:)

1. The interpreter takes care of expanding blank suppression codes (DLE's), EOF (the end of file character), and adding line feeds to every carriage return.
2. PRINTER: read,write and init have only the return address on the stack. PRINTER: status has the same items on the stack as CONSOLE: status. PRINTER: init should cause the PRINTER: to do a carriage return and a line feed and throw away any characters buffered to be printed. No form feed should be done.
3. For status, return in the first word of the status record the number of bytes buffered in the direction asked for; if this cannot be determined by your PRINTER:, return a 0.
4. The PRINTER: write routine must buffer a line and send it all at once if your PRINTER: can only receive data that way.
5. Line Delimiter characters:

A. CR (hex 0D) A carriage return should cause the PRINTER: to print the current line and return the carriage to the first column. An automatic line feed should not be done by the PRINTER: driver when it reads a CR.

B. LF (hex 0A) The RSP will send line feeds to the PRINTER: driver after each carriage return. This should cause the PRINTER: to advance to the next line. If the PRINTER: must also do a carriage return when it is given a line feed, then this is O.K.

C. FF (hex 0C) This should cause the PRINTER: to move the paper to top of form and do a carriage return. If top of form is not possible on your PRINTER:, do a carriage return followed by a line feed.

6. It is assumed that input cannot be received from the PRINTER:. See the BIOS section for a discussion of how to get input from the PRINTER:. Normally, trying to get input from the PRINTER: should return completion error code #3.

Units 7 (REMOTE: in) & 8 (REMOTE: out)

1. These must both go to the same driver.
2. The interpreter takes care of expanding blank suppression codes (DLE's), EOF and adding line feeds to every carriage return.
3. Same stack setup as the PRINTER:.
4. Status should return in first word of status vector the number of bytes buffered for the direction specified in the control word, 0 if you have no way to check.
5. This unit is supposed to be an RS-232 serial line for many different applications so it is necessary that it transfer the data without modifying it in any way. The transfer rate default is 9600 baud.
6. It would be nice if the input to REMOTE: could be buffered in the same way input to the CONSOLE: is but this is not an absolute requirement.
7. REMOTE: init should set up the REMOTE: device so it is ready to read and write.

B. Block Structured Devices

Units 4 (the boot unit),5,9,10,11,12.

1. These units are assumed to be block structured devices, the drivers for these units must do their own Pascal Block to Track-Sector conversions.
   The UCSD system assumes the disk device is a 0-based consecutive array of 512 byte logical blocks. All UCSD Pascal disks must have this logical structure no matter what their actual physical structure or size are. The physical allocation schemes for information on different types of disks are arranged with sectors that are of various sizes that depend on the hardware of the particular disk device used. The driver must convert the Pascal block # to the appropriate track & sector # of where that block is stored on it's disk device. This could be a floppy or hard disk or some other type of device. It doesn't really matter, so long as your driver maps the Pascal Block to the cOrrect place and continues to do so for the length (byte count) required for the UnitIO operation.
   The Pascal system uses logical blocks 0 & 1 for it's bootstrap code. These logical blocks should not be used for anything else and should therefore only be available to Pascal through direct UNITREAD & UNITWRITE operations and not accessable by the system through any other means. This document will not attempt to describe the boot sequence & does not attempt to give you enough information to attach another driver or device to unit #4: so you can cold boot from that unit.
   When a UNITWRITE is done to disk where the byte count MOD 512 is not equal to 0 (this means the last block included in the write would be partially written to according to the byte count), it is undefined whether garbage is written into the remaining part of this last block. So you may write a whole block anyhow if that is more efficient and the Pascal system will not suffer any bad consequences.
   When a UNITREAD is done from a disk you are not allowed to overwrite into the unused part of the last block (if there is an unused part due to byte count MOD 512 <> 0). You must only

send the number of bytes asked for because you could clobber memory having some other valid use if you wrote extra bytes. You will have to buffer the last sector inside your disk read routine then transfer exactly the number of bytes from this last sector needed to add up to the total bytes requested.

2. The unit number will always be in the A register.
3. The stack setup for read and write is:

```
CONTROL WORD (The MODE parameter mentioned in the
                    1.1 Language Ref Manual on page 41)
DRIVE NUMBER
BUFFER ADDRESS
BYTE COUNT
BLOCK NUMBER
RETURN ADDRESS <--TOS
```

For init there is only the return address on the stack and for status the setup is the same as for the CONSOLE:.

4. Status requests should return the following in the status record:

```
word1:Number of bytes buffered in the direction asked
      for in the control word. Return 0 if you have no
      way of checking.
word2:Number of bytes per sector.
word3:Number of sectors per track.
word4:Number of tracks per disk.
```

C. Other

Unit 3

1. This unit has no meaning for the Apple II system except that UNITCLEAR on this unit sets text mode.

Considerations when attaching drivers for user defined devices numbers 128-143.

These unit numbers are provided for you to do whatever you want with them. you can define what they do except for the following protocols.

1. Follow the considerations for all drivers listed above.
2. The unit number will always be in the A register.
3. The stack setup for read and write is:

```
CONTROL WORD
DRIVE NUMBER
BUFFER ADDRESS
BYTE COUNT
BLOCK NUMBER
RETURN ADDRESS <--TOS
```

For init there is only the return address on the stack and for status the setup is the same as for the CONSOLE:.

This is a sample driver for a user defined device.

```
        ;Locations 0..35 hex may be used as pure temps. One should
        ;never assume these locations won't be clobbered if you leave
        ;the environment of the driver itself. ("leaving" includes
        ;calls to CONCK).

CONCKADR .EQU 02

        ;Only one .PROC may occur in a driver, each driver to be
        ;ATTACHED must be assembled separately using the Pascal
        ;assembler and can have no external references.

        .PROC U128DR

        STA TEMP1 ;Save Areg contents (unit#)
        PLA
        STA RETURN
        PLA
        STA RETURN+1
        TXA ;Use the X reg to tell you what kind of
                ;call this is.
        CMP #2
        BEQ INIT
        CMP #4
        BEQ STATUS
        CMP #0
        BEQ PMS
        CMP #1
        BEQ PMS

        ;Could have error code here
        JMP RET

PMS PLA ;Get the parameters
        STA BLKNUM
        PLA
        STA BLKNUM+1
        PLA
        STA BYTECNT
        PLA
        STA BYTECNT+1
        PLA
        STA BUFADR
        PLA
        STA BUFADR+1
        PLA
        STA UNITNUM ;Also in TEMP1
        PLA
        STA UNITNUM+1 ;Should always be 0
        PLA
        STA CONTROL
        PLA
        STA CONTROL+1
        TXA
        BNE WRITE
```

```
READ JSR GOTOCK
         ;Your driver's code for a read
         (If more than one unit were attached to this driver, this
          code could jump to various places depending on the contents
          of the Areg stored in TEMP1)
         JMP RET

WRITE JSR GOTOCK
         ;Your driver's code for a write
         JMP RET

         ;If you wanted to call CONCK whenever your device did a read
         ;or write, you would use this routine:
CKR .WORD CONCKRTN-1
GOTOCK LDY #55. ;Offset to address of CONCK
         LDA @0E2,Y
         STA CONCKADR
         INY
         LDA @0E2,Y
         STA CONCKADR+1
         LDA CKR+1 ;Set it up so you return to CONCKRTN after
         PHA ;the CONCK call.
         LDA CKR
         PHA
         JMP @CONCKADR ;Jump to CONCK
CONCKRTN RTS ;Return to caller.

INIT ;Your driver's code for init
         JMP RET

STATUS PLA
         STA CONTROL
         PLA
         STA CONTROL+1
         PLA
         STA BUFADR ;Address of status record.
         PLA
         STA BUFADR+1
         ;Your driver's code for status

RET LDA RETURN+1
         PHA
         LDA RETURN
         PHA
         LDA TEMP1
         RTS


RETURN .WORD 0 ;Can't use 0 page for these since we leave
TEMP1 .WORD 0 ;our environment when going to CONCK.
CONTROL .WORD 0
UNITNUM .WORD 0
BUFADR .WORD 0
BYTECNT .WORD 0
BLKNUM .WORD 0
```

```
        .END
```

This is a sample driver for a CONSOLE: driver replacement.

```
ROUTINE .EQU 02
TEMP1 .EQU 04


        .PROC CKATCH

        JMP CONCKHDL ;SYSTEM.ATTACH will patch the start of CONCK
                     ;to jump here when you attach a driver to the
                     ;CONSOLE:.

                     ;We are not popping the return address from
                     ;the stack cause we'll return from the system
                     ;routine we call from this driver.
        STA TEMP1 ;All the read,write,init and stat calls will
                     ;jump here (the starting address of your
                     ;CONSOLE: driver+3).

        STY TEMP1+1
        TXA
                     ;This example shows you how to have your
                     ;own code for the CONSOLE: as well as using
                     ;the system CONSOLE: routines. If you want
                     ;to replace the system routines completely,
                     ;you need to pull the return address here.
        BEQ READ
        CMP #1
        BEQ WRITE
        CMP #2
        BEQ INIT
        CMP #4
        BEQ STATUS

        ;Error code here

READ ;Your driver's code for a read

        LDY #1 ;offset to address of CONSOLE: read in
                     ;the copy of the jmp vector made by
                     ;SYSTEM.ATTACH. See the jump vectors in the
                     ;BIOS section below to see how we get   e
                     ;offsets.

        BNE GET

        ;You would have a JMP RET here (see example for user defined
        ;device) if you were not using the system CONSOLE: routines
        ;as well.
```

```
WRITE ;Your driver's code for a write
        LDY #4
        BNE GET

INIT ;Your driver's code for init
        LDY #7
        BNE GET

STATUS ;Your driver's code for status
        LDY #43.

GET LDA @0E2,Y ;At E2 is a pointer to the copy of the
                        ;jump vector made by SYSTEM.ATTACH before
                        ;it was modified to attach your drivers.

        STA ROUTINE
        INY
        LDA @0E2,Y
        STA ROUTINE+1
        LDY TEMP1+1 ;Restore registers
        LDA TEMP1
        JMP @ROUTINE ;Go to the original CONSOLE: driver for this
                        ;I/O command. You will return from there; the
                        ;BIOS is already folded in due to the way your
                        ;driver was attached by SYSTEM.ATTACH.


CONCKHDL PHP ;Duplicate the 1st three instructions of CONCK
        PHA ;as they were patched by SYSTEM.ATTACH to jump
        ;TXA below ;to the 1st instruction of this driver.

        ;Here you can put the code for your own part of CONCK (you
        ;may want to check some additional device like a keypad or
        ;something or you may want to replace the system CONCK
        ;routine alltogether. If you do this, you must save the rest
        ;of the machine state and return it when you are finished.
        ;See example below.

        TYA ;Save Yreg contents for a second.
        PHA

        ;This code gets us to the system CONCK routine.
        CLC
        LDY #55. ;Offset to the address of system CONCK in the
                        ;copy of the original jmp vector.

        LDA @0E2,Y
        ADC #3 ;Add 3 so you enter right after the three
                        ;instructions you duplicated at CONCKHDL.
        STA ROUTINE
        INY
        LDA @0E2,Y
        ADC #0
        STA ROUTINE+1
        PLA ;Restore Yreg.
```

```
        TAY
        TXA ;Last of CONCK instructions SYSTEM.ATTACH
                    ;overwrote with the jmp to the start of this
                    ;driver.

        JMP @ROUTINE ;Goto system CONCK and return from there.

        .END
```

Here is another alternative for the CONCKHDL part of the above program.

```
CKRTN .WORD CONCKRTN-1
CONCKHDL ; 1.If you don't care about type-ahead, this could be
         ; simply the following code (assuming your CONSOLE:
         ; read gets a character directly from your CONSOLE:
         ; device whenever it is called) :

-----------------------------------------------------------------
         PHP
         INC RANDL ;RANDL is a permanent word at BF13 used in
                   ;the built in random function.
         BNE $1
         INC RANDH ;RANDH
$1 PLP
         RTS
-----------------------------------------------------------------

         ; 2.If you want type-ahead, this code should check to see
         ;if there is a character available and stuff it into a type-
         ;ahead buffer.
         ; 3.If you are using this with the regular CONCK (extra keypad
         ;to check or statistics for example), then you can do it this
         ;way.

-----------------------------------------------------------------
         PHP ;Save state of machine
         PHA
         TXA
         PHA
         TYA
         PHA

         ;Put your driver's part of CONCK here (gives your driver
         ;priority)

         LDA CKRTN+1 ;Set up things to return from reg CONCK
         PHA
         LDA CKRTN
         PHA
         PHA ;Push garbage to account for other pushes done
         PHA ;in first three bytes of CONCK

         CLC ;Setup to call CONCK
         LDY #55. ;Offset to the address of system CONCK in the
```

```
                    ;copy of the original jmp vector.

        LDA @0E2,Y
        ADC #3 ;Add 3 so you enter right after the three
                    ;instructions you duplicated at CONCKHDL.
        STA ROUTINE
        INY
        LDA @0E2,Y
        ADC #0
        STA ROUTINE+1
                    ;In this example we don't have to worry about
                    ;the machine state here as we are restoring
                    ;it after we call CONCK

        JMP @ROUTINE ;Goto system CONCK and return to CONCKRTN

CONCKRTN PLA ;Restore state of machine
        TAY
        PLA
        TAX
        PLA
        PLP
        RTS ;Return to the guy who called CONCK.
```

MAKING ATTACH.DRIVERS

1. Xecute the standard 1.1 LIBRARY program.
2. The output code file should be ATTACH.DRIVERS or could be named somethine else and renamed ATTACH.DRIVERS when you put it on the boot disk.
3. For the Link code file use the code file of your first driver.
4. Copy its slot #1 into slot #0 of ATTACH.DRIVERS.
5. As long as you have more drivers to add, use N(EW to get another Link code file and copy it's slot #1 into slots #2,3,...15 of ATTACH.DRIVERS.
6. When done, type 'Q' then 'N' followed by a RETURN for the notice. See the 1.1 Operating System Reference Manual for further info on the LIBRARY program.

THE WORKINGS OF SYSTEM.ATTACH

If it is on the boot disk, SYSTEM.ATTACH is Xecuted by the operating system (both regular 1.1 and runtime 1.1) before SYSTEM.STARTUP. The 1.1 runtime system will use a runtime version of SYSTEM.ATTACH.

The error messages that can be generated by SYSTEM.ATTACH are:

1. ERROR =>No records in ATTACH.DATA
2. ERROR =>Reading segment dictionary of ATTACH.DRIVERS
3. ERROR =>reading driver
4. ERROR =>A needed driver is not in ATTACH.DRIVERS
5. ERROR =>ATTACH.DATA needed by SYSTEM.ATTACH

6. ERROR =>ATTACH.DRIVERS needed by SYSTEM.ATTACH

If all goes well attaching drivers, SYSTEM.ATTACH will display nothing unusual in the regular boot sequence except for extra disk accesses and anything done in the init calls to any of the attached devices.

II.BIOS

This section explains things in the BIOS area that are extensions and modifications that were added to Apple Pascal version 1.1 that were different or not there at all in Apple Pascal version 1.0 (UCSD version II.1).

1. The disk routines have been modified to handle interrupts (So interrupt driven devices could be attached to 1.1 Pascal) if they are being used. To use interrupts, one would have to attach an interrupt driver, then patch the IRQ vector (FFFE hex) to point to this driver. The Pascal system is defined to come up with interrupts turned off so, once the driver is brought in and the IRQ patched, interrupts must be turned on. The driver's init call could patch the IRQ and turn on interrupts. The disk routines save the current state of the system and turn interrupts off only during crucial time periods, the state of the system is returned during non crucial time periods so interrupts can be handled. This has not been tested at this time, so there is no data concerning the maximum interrupt response time delay.

2. The control word parameter in UNITREAD and UNITWRITE was not passed on to the BIOS level routines from the RSP level. This has been done in 1.1 to allow the changes to the control word listed below under special character checking and also so user defined units or attached Pascal units can use the user defined bits of the control word.

3. IORESULTS 128-255 are available for user definition on user defined devices.

4. UNITSTATUS has been implemented in the Apple II Pascal 1.1 system. This works for the Pascal system units as described in the ATTACH part of this document. For user defined units, Unitstatus can be used for whatever necessary.

   Unitstatus is a procedure that can be called from the Pascal level in the same way Unitread can. It has three parameters:

   1. unit#.
   2. pointer to a buffer.
      (any size buffer you want of type Packed Array of Char)
   3. control word.

When you make a Unitstatus call from Pascal, the call should look like:

```
UNITSTATUS(UNITNUM,PAC,CONTROL);
```

Where UNITNUM & CONTROL are integers and PAC is a Packed Array of CHAR or a STRING and may be subscripted to indicate a starting position to transfer data to or from. See further information on what Unitstatus is defined to do for the various devices in the ATTACH part of this document.

The control word will tell the status procedure for a particular unit what information about the unit you want. Bit 0 of this word should equal 1 for input status and 0 for output status. Unitstatus is implemented with bit 1 of the control word =1 meaning the call is for unit control. When this bit =0 the call is for unitstatus. In all cases bits 2-12 are reserved for system use and bits 13-15 are available for user defined funtions.

An entry in the jump vector has been made for each of the system Unitstatus calls, i.e. CONSOLESTAT,PRINTERSTAT,REMOTESTAT,etc.. Unitstatus calls to a user defined device (128-143) will all go through the same jump vector location.

5. The handling of CTRL-C by the Apple bios was non standard in 1.0. The UCSD BIOS definition specifies that a CTRL-C coming from REMOTE: or the PRINTER: should be placed in the input buffer and then no more characters should be received. Our bios did fill the buffer with nulls including the place where the CTRL-C was to go. Apple Pascal's BIOS now conforms to the standard definition, where the null filling of the buffer is done only when CTRL-C comes from the CONSOLE: (#1:).

6. The unitio routines can be accessed from assembly procedures by pushing the correct parameters on the stack and using the jump vector to get to the BIOS routine. A seperate document needs to be written describing how this is done and pointing out the problems doing it in the case of the CONSOLE:,SYSTERM:,PRINTER: & REMOTE: units. These problems are concerned with the special character handling done in the RSP for these units. The assembly procedures calling the pascal drivers for these units would either have to repeat portions of the RSP code themselves or not get the special character handling provided by the RSP. Calling the CONSOLE: init routine requires pointers to syscom and the break routine to be passed on the stack. These pointers are now stored in a fixed location so assembly routines wanting to call coninit can get at them. See the locations section.

7. Suppression of Special Character Checking.

   Special characters in the Pascal system are of three types:

   A. Chars used to control the 40 character screen. These are ctrl-A,Z,W,E & K.

   B. Pascal system control chars for general CONSOLE: use. These are ctrl-S & F.

C. Types A & B are checked for by the CONCK funtion in the bios. There are other special chars checked for in the RSP. These are ctrl-C, DLE, and CR (line feeds are automatically appended to CR). With UNITREAD and UNITWRITE the automatic handling done by the Pascal system of these characters can be turned off. To turn off DLE expansion and EOF checking give bit 2 of the control word a value of 1. The automatic adding of line feeds to carriage returns can be suppressed by setting bit 3 of the control word to 1.

A way was needed to suppress special handling for types 'A'&'B'. This can now be done in two ways. First, the control word of UNITR/W will turn off checking for type 'A' control chars if bit 4 is set and will turn off checking for type 'B' chars if bit 5 is set. In this mode, the special char handling will only be turned off during that particular unitio. This will be be done for you in the RSP by setting bits in a byte 'SPCHAR' at location BF1C. The CONCK routine will look at bit 0 of SPCHAR and if set will not look for the type 'A' chars; if bit 1 is set, it will not look for the type 'B' chars. If you set these bits in the SPCHAR yourself instead of letting the RSP do it through the unitio control word, then the associated special character checking will be turned off until you reboot or reset the bits again. When special char checking is turned off, the chars are passed back to the Pascal level like all other chars would be. You can use these added features to redefine the system special chars in a particular application program or to just disable them.

8. The EOF char (ctrl-C) causes a lot of problems in the Pascal system. The cause of the problems is that the editor looks for this character to end many of it's editing modes. The editor has it's own getchar routine which reads each character the user enters from SYSTERM:. When reading from SYSTERM: instead of the CONSOLE:, the EOF char is passed back as any other character but it still ends the current call to unitread. The editor echoes each char to the CONSOLE: itself until it comes to ctrl-C. The operating system and the filer both use the getchar routine in the operating system. This routine is defined to re-init the system if it gets a ctrl-C from the CONSOLE: and it reads from the CONSOLE:, not SYSTERM:. You must be sure not to end responses with control-C except for the cases (in the editor only) that are supposed to end with control-C. See the 1.1 Operating System Reference Manual.

9. The bios card recognizing section has been enhanced to recognize a new 'FIRMWARE' type card. This card will allow OEM's to have their drivers in their own firmware on the card. Routines have been added to allow for init,read,write & status calls to this new type card. This protocol has been documented and is attached as an appendix to this document.

10. As you can see, the Pascal system memory usage is scattered all over the 64k space. The Apple II was not designed with a stack machine, like the Pascal P-machine, in mind. We don't need any more constraints fixing certain pieces of the system

to certain EXACT places. To make the best use of the space we have, we must have the ability to move things around. To achieve this goal, we intend the following:

A. To stop people from writing things that peek here and poke there and expect things to stay exactly where they were for future versions.

B. Various people need space for patch areas and other purposes. All programs have to be written so this space does not have to be in a permanent fixed location if this is at all possible. The areas reserved for system use are filling up fast, we need to avoid using them. You can get space dynamically using NEW but you must be careful that this space stays around for the whole time you need it. If you are attaching a driver, you can get buffer space in the driver by using .WORD or .BLOCK in the Assembler. This space can be accessed from outside the driver if you know the offset to the start of this space from the start of the driver. This method could even be used to get space below the heap by attaching a driver to one of the user defined devices that is a large .BLOCK and is only used as a buffer. You can get the address of this buffer (of a driver) from the jump vector that has a pointer to the driver. Pointers to all the jump vectors are in zero page, see the locations section below.

C. The jump vector will have a fixed order for version 1.1 and future versions. The order is the same as in the old version 1.0 with the new entrys added to the bottom. The setup for the jump vector and getting into the BIOS is different than the old 1.0 system. Here is how the new system is set up with the fixed order for the jump vector:

```
;-------------------------------------------------
;
; MAIN BIOS JUMP TABLE CALLED FROM INTERPRETER
; (FOLLOWED BY REAL JUMP TABLE AT FIXED OFFSET)
; RSP CALLS COME TO THIS JUMP VECTOR
;
;-------------------------------------------------
BIOS JSR SAVERET ;CONSOLE READ ;Jump vector before fold.
          JSR SAVERET ;CONSOLE WRITE
          JSR SAVERET ;CONSOLE INIT
          JSR SAVERET ;PRINTER WRITE
          JSR SAVERET ;PRINTER INIT
          JSR SAVERET ;DISK WRITE
          JSR SAVERET ;DISK READ
          JSR SAVERET ;DISK INIT
          JSR SAVERET ;REMOTE READ
          JSR SAVERET ;REMOTE WRITE
          JSR SAVERET ;REMOTE INIT
          JSR SAVERET ;GRAFIC WRITE
          JSR SAVERET ;GRAFIC INIT
          JSR SAVERET ;PRINTER READ
          JSR SAVERET ;CONSOLE STAT
          JSR SAVERET ;PRINTER STAT
```

```
           JSR SAVERET ;DISK STAT
           JSR SAVERET ;REMOTE STAT
CONCK JSR SAVERET ;To get to CONCK from CONCKVEC
           JSR SAVERET ;USER READ For UDRWIS
                       ;USER WRITE
                       ;USER INIT
                       ;USER STAT
            JSR SAVERET ;For PSUBDR
            JSR SAVERET ;IDSEARCH
               .
               .
               .
```

```
 THIS JUMP TABLE MUST BE OFFSET
 FROM BIOSTBL BY EXACTLY $5C.
 SYSTEM.ATTACH MODIFYS THIS JUMP
 VECTOR TO POINT TO ATTACHED DRIVERS
 FOR THE STANDARD SYSTEM UNITS.
```

```
IOSAF JMP CREAD ;Jump vector after fold.
           JMP CWRITE
           JMP CINIT
           JMP PWRITE
           JMP PINIT
           JMP DWRITE
           JMP DREAD
           JMP DINIT
           JMP RREAD
           JMP RWRITE
           JMP RINIT
           JMP IORTS ;Do nothing for GRAFWRITE.
           JMP GRAFINIT
           JMP IORTS ;Do nothing for PRINTER: read.
           JMP CSTAT
           JMP ZEROSTAT ;For PRINTER: stat, pop params & store 0
                        ;in 1st buffer word.
           JMP DSTATT
           JMP ZEROSTAT ;For REMOTE: stat, pop params & store 0
                        ;in 1st buffer word.
           JMP CONCK
           JMP UDRWIS ;Routine to get to user defined devices, see
                        ;ATTACH part of document for description of
                        ;this routine.
           JMP PSUBDR ;Routine to get to drivers that are substituted
                        ;for the standard Pascal disk units 4,5,9..12.
                        ;See ATTACH part of document for description of
                        ;this routine.
           JMP IDS
```

```
; STRIP LOCAL RETURN ADDR,
; STRIP PASCAL ADDR AND SAVE IN RETL,RETH
; PLACE 'GOBACK' ON RETURN STACK
; THEN RESTORE LOCAL RET ADDR & RETURN
; MEANWHILE UNFOLD BIOS INTO DXXX
;
;---------------------------------------
SAVERET STA TT1 ;SAVE A REG
            PLA
            CLC
            ADC #05A ;ADD OFFSET TO JUMP TABLE (BIOSAF)
            STA TT2 ;LOCAL RET ADDR
            PLA
            ADC #0
            STA TT3
            PLA
            STA RETL ;PRESERVE PASCAL RETURN
            PLA
            STA RETH
            .IF RUNTIME=0
            LDA 0C083 ;UNFOLD BIOS INTO DXXX
            .ENDC
            LDA TT1 ;RESTORE A-REG
            JSR SAVRET2 ;PUTS 'GOBACK' ON STACK

;---------------------------------------
;
; FOLD INTERP INTO DXXX
; THEN RETURN TO PASCAL VIA
; RETURN ADDR SAVED IN RETL,RETH
;
;---------------------------------------
GOBACK STA TT1 ;SAVE A-REG
            LDA RETH
            PHA
            LDA RETL
            PHA
            .IF RUNTIME=0
            LDA 0C08B ;FOLD INTERP INTO DXXX
            .ENDC
            LDA TT1
            RTS ;AND BACK TO PASCAL

SAVRET2 JMP @TT2 ;JUMP INTO JUMP TABLE (BIOSAF)
```

D. In zero page are two words pointing to the base of the two jump vectors (before and after the fold). These are stored in PERMANENT locations that had a value of 0 in the old 1.0 release and were not used by the system (see locations section). Applications needing to patch the jump vectors can store the offset from the vector base in the Y reg and use indirect indexed addressing to do the patch. The application will need to have the vector base locations for the old release hardcoded in as the base pointer for the old 1.0 release will

be 0. If you want to write an application that works with 1.0 and 1.1 and future versions, you know if the zero page vector pointers are 0 it's the 1.0 system otherwise it's 1.1 or a future version which will use the same protocols as 1.1 as described in this document.

It is important that any application patching the jump vector temporarily then returning it to its original value get the original value from the vector itself before the patch and put it in a storage location. When the vector needs to be restored to it's original state, use this storage location for it's original value. The patches should be done in this manner so the applications doing the patches will always return the system to it's original state no matter what past, present or future Pascal version it is patching.

E. For CONSOLE: init to be used from assembly routines the locations of SYSCOM and the BREAK routine have to be available. The CONINIT routine requires these on the stack. Pointers to SYSCOM and BREAK will be stored by the interpreter boot in a PERMANENT location in the BF00 page (see locations section).

F. Since the old 1.0 release, the code to jump to the CONCK routine has been set up at location BF0A. Anyone wishing to get to the CONCK routine should do a JSR BF0A as this will always get them there no matter where the CONCK routine really is. The keypress function has been changed to conform to this new convention but it will use the old convention if it is working from within an old system. Do not try to get to CONCK in this way from within an ATTACHED driver as you will loose your return address to Pascal. See ATTACH part of this document for how to get to CONCK from an attached driver.

G. There is now a version byte so one can tell which version (1.0, 1.1, etc.) of Apple Pascal he is working with. There is also a flavor byte to tell one which flavor of this version he has (regular, runtime, runtime without sets, etc.). (see locations section)

11. Whenever SYSTEM.ATTACH is used, it will make a copy of the original BIOS jump vector (the after fold vector that has the actual driver addresses in it) and put this below the heap with the drivers that are attached. It will leave a pointer to this copy of the vector at location 00E2. You can use this vector in you drivers to get to the standard Apple drivers for any device. This way you can define a driver that does something above and beyond the standard Apple driver yet this new driver can still make use of the standard Apple driver. See the ATTACH part of this document for more information.

12. In the RSP are two vectors that tell the RSP what is legal (input &-or output) for a particular character orientated device (CONSOLE:, REMOTE: & PRINTER:). For example it tells the RSP that it is illegal to read from the PRINTER:. If you wanted to ATTACH a PRINTER: driver so you could read from the PRINTER:, you would have to change this vector. 00E4 points to the READTBL vector

and 00E6 to the WRITTBL vector. Let's take the READTBL for an example:

```
READTBL ;table of routine addresses to be called when
                ;writing to that unit (disk I/O does not use
                ;this table).
                ;an entry=0 means that the operation is illegal
                ;for that unit.
                .WORD BIOS+CONREAD ;unit 1
                .WORD BIOS+CONREAD ;unit 2
                .WORD 0 ;unit 3
                .WORD 0 ;4 & 5 are disk units
                .WORD 0
                .WORD 0 ;6 is PRINTER:
                .WORD BIOS+REMREAD ;unit 7
                .WORD 0 ;8 is rem write which has
                                        ;an address in the WRITTBL
```

Here BIOS refers to the base of the jump vector before the fold and CONREAD is the offset off the base of that vector to get to the jump to the CONSOLE: read routine (for CONSOLE: read the offset is 0, for CONSOLE: write it's 3, etc). The value for BIOS is the pointer stored in location 00EC mentioned in the locations section below.

LOCATIONS.

These are the locations of new system permanents mentioned in this document, all pointers are set up by the system and are stored low byte first. Do not modify what is stored in these pointers (except for SPCHAR if you want to suppress special character checking) since the system uses this information too. These locations are defined to have the same function & remain in the same place for future versions of Apple II Pascal.

```
BF1C SPCHAR (To control special chars)

BF1D IBREAK (Set by boot in interp for assembly calls to CONINIT)
BF1F ISYSCOM ( '' )

BF21 VERSION (1 byte Version # of system, =2 for the new release, 0
                    for the old 1.0 release)
BF22 FLAVOR (This byte tells which flavor [runtime,regular,
                    etc.] of this VERSION you are dealing with)
                    The encoding is:
                                    1 -->regular system
                                         runtime versions:
                                    2 -->LC-ALL (LC- means no
                                          language card)
                                    3 -->LC-no sets
                                    4 -->LC-no floating point
                                    5 -->LC-no sets or floating point
                                    6 -->LC+ALL
                                    7 -->LC+no sets
```

8 -->LC+no floating point
9 -->LC+no sets or floating point
This flavor byte is 0 in the old 1.0 release.

BFC0-BFFF BDEVBUF (Area for non Apple boot devices, like the CORVUS)
00E2 ACJVAFLD (Pointer to ATTACH copy of the original Jump Vector after the fold)
00E4 RTPTR (Pointer to READTBL)
00E6 WTPTR (Pointer to WRITTBL)
00E8 UDJVP (Pointer to user device jump vector)
00EA DISKNUMP (Pointer to disknum vector)
00EC JVBFOLD (Pointer to jump vector before fold)
00EE JVAFOLD (Pointer to jump vector after fold)

FFF6 (Version word which = 1 for version 1.0 and
= 0 for version 1.1
This version word should not be used at runtime to tell which version you have. For that use the version byte mentioned above. This word should only be used by software that wants to see which SYSTEM.APPLE it is dealing with by looking at the contents of this word in the SYSTEM.APPLE file when it is not loaded in memory)
FFF8 (Start vector)
FFFA (NMI non maskable interrupt vector)
FFFC (RESET vector)
FFFE (IRQ interrupt request vector)

The locations and code in the 1.0 'PRELIMINARY APPLE PASCAL GUIDE TO INTERFACING FOREIGN HARDWARE' BIOS document are not the same for Apple Pascal 1.1 and that document clearly stated we would not commit ourselves to keeping them the same.

Pascal 1.1 Firmware Card Protocol

One major problem with Apple Pascal 1.0 is the way it deals with peripheral cards. It was set up to work with the four peripheral cards that Apple supported at the time of its release (the disk,communciations,serial and parallel cards) and had no mechanism for interfacing any other devices. Since Apple as well as many other vendors continue to produce new peripherals for the Apple][, a new protocol was designed and implemented in the Pascal 1.1 BIOS which allows new peripheral cards to be introduced to the system in a consistent and transparent fashion. The new protocol is called the "firmware card" protocol since the BIOS deals with these cards by making calls to their firmware at entry points defined by a branch table on the card itself. The new protocol fully supports the Pascal typeahead function and KEYPRESS will work with firmware cards used as CONSOLE devices. The following paragraphs describe the firmware card protocol in full detail.

A firmware card may be uniquely identified by a four byte sequence in the card's $CN00 ROM space. Location $CN05 must contain the value $38 and location $CN07 must contain $18. Note that these are identical to the Apple

Serial Card. A firmware card is distinguished from a serial card by the further requirement that location $CN0B must contain the value $01. This value is called the "generic signature" since it is common to all firmware cards. The value at the next sequential location, $CN0C, is called the "device signature" since it uniquely identifies the device.

The device signature byte is encoded in a meaningful way. The high order 4 bits specify the class of the device while the low order four bits contain a unique number to distinguish between specific devices of the same class. The appendix to this document defines some device class numbers; in any case vendors should contact Apple Technical Support to make sure they use a unique number for their device signature. Although the device signature is ignored by the 1.1 BIOS, it may be used by applications programs to identify specific devices.

Following the 2 signature bytes is a list of four entry point offsets starting at address $CN0D. These four entry points must be supported by all firmware cards. They are the initialization, read, write and status calls. The BIOS takes care of disabling the $C800 ROM space of all other cards before calling the firmware routines.

The offset to the initialization routine is at location $CN0D. Thus, if $CN0D contains XX, the BIOS will call $CNXX to initialize the card. On entry, the X register contains $CN (where N is the slot number) and the Y register contains $N0. On exit, the X register should contain an error code, which should be 0 if there was no error. This error code is passed on to the higher levels of the system in the global variable "IORESULT". Registers do not have to be preserved.

The offset to the read routine is at location $CN0E. On entry, the X register will contain $CN and the Y register will contain $N0. On exit, the A register should contain the character that was read while the X register contains the IORESULT error code. The A and Y registers do not have to be preserved.

The offset to the write routine is at location $CN0F. On entry, the A register contains the character to be written while the X register contains $CN and the Y register contains $N0. On exit the X register should contain the IORESULT error code (which should be 0 for no error). The A and Y registers do not have to be preserved.

The offset to the status routine is at location $CN10. On entry, the X register contains $CN and the Y register contains $N0 while the A register contains a request code. If the A register contains 0, the request is "are you ready to accept output?". If the A register contains 1, the request is "do you have input ready for me?". On exit, the driver returns the IORESULT error code in the X register and the results of the status request in the carry bit. The carry clear means "false" (i.e., no, I don't have any input for you), while the carry set means true. Note that the status call must preserve the Y register but does not have to preserve the A register.

Thus, sample code for the first few bytes of a firmware card's $CN00 space should look something like:

```
    BASICINIT BIT $FF58 ;set the v-flag
```

```
                BVS BASICENTRY ;always taken
IENTRY SEC ;BASIC input entry point
                DFB $90 ;opcode for BCC
OENTRY CLC ;BASIC output entry point
                CLV
                BVC BASICENTRY ;Always taken
;
; Here is the Pascal 1.1 Firmware Card Protocol Table
;
                DFB $01 ;Generic signature byte
                DFB $41 ;Device signature bye
;
PASCALINIT DFB >PINIT ; > means low order byte
PASCALREAD DFB >PREAD ;offset to read
PASCALWRITE DFB >PWRITE ;offset to write
PASCALSTATUS DFB >PSTATUS ;offset to status routine
```

The above code fulfils all the requirements for both the BASIC and Pascal 1.1 I/O protocols. The routines PINIT, PREAD, etc, are probably jumps into the card's $C800 space which is already properly enabled by the BIOS. The reason the $CN00 space was chosen for the protocol (as opposed to the $C800 space) is that the BASIC protocol requires that all cards have $CN00 ROM space while some smaller cards may not need any $C800 ROM space.

The firware card protocol includes 2 optional calls that do not have to be implemented but would be kind of nice. The BIOS checks location $CN11 to determine if the optional calls are present; if that location contains a $00 then the BIOS thinks the calls are implemented. Thus if your card does not implement the optional calls, you should ensure that $CN11 contains a non-zero value. The two optional calls are a control call pointed to by $CN12 and an interrupt handler call pointed to by $CN13.

The control call entry point is specified by the offset at $CN12. On entry, the X register contains $CN, the Y register contains $N0 and the A register contains the control request code. Control requests are defined by the device. On exit the X register should contain the IORESULT error code.

The interrupt poll entry point is specified by the offset at $CN13. On entry, the X register contains $CN and the Y register contains $N0. The interrupt poll routine should poll the card's hardware to determine if it has a pending interrupt; if it does not it should return with the carry clear. If it does, it should handle the interrupt (including disabling it) and return with the carry set. Also, the X register should contain the IORESULT error code which should be 0 if there was no error. An interrupt polling routine must be careful not to clobber any zero page or screen space temporaries.

The control and interrupt requests are not implemented in the Pascal 1.1 BIOS but it would be nice to support them if possible as they may be implemented in later versions of the Pascal BIOS as well as other forthcoming operating system environments for the Apple][.

Note that the firmware card signature is a superset of the Apple serial card signature as recognized by the Pascal 1.0 BIOS. This allows a firmware card to function with both Pascal 1.0 and Pascal 1.1. If a card wishes to

work with Pascal 1.0 as a "fake" seral card, it must provide an input entry point at $C84D and an output entry point at $C9AA. Note that since Pascal 1.0 will think the card is a serial card, typeahead and KEYPRESS capabilities will be lost.

Additional Notes

1. The Pascal RSP expects the high order bit of every ASCII character it receives from the Console read routine to be clear. The RSP will not do this for you; you must ensure the high bit of all text your card passes to the RSP from the console read routine is clear.

2. Zero page locations $00 to $35 may be used as temporaries by your firmware, as are the slot 0 screen space locations ($478,$4F8, etc.). In general, peripheral card firmware should be as conservative as possible in their memory usage, preserving zero page contents whenever possible. An interrupt polling routine must not destroy these or any other memory locations.

3. Location $7F8 must be set up to contain the value $CN, where N is the slot number, if your card utilizes the $C800 expansion ROM space. The BIOS does not do this for you; his must be done if you want your card to function in an interrupting environment.

4. The firmware card status routine should be as quick as possible, as it may be called from within the I/O polling loops of many other peripherals if your card is being used as the console device. In no case should the status routine take longer than 100 milliseconds.

5. A firmware card in slot 1 is automatically recognized as the volume "PRINTER:". A firmware card in slot 2 is automatically recognized as the volumes "REMIN:" and "REMOUT:". A firmware card in slot 3 is automactically recognized as the volumes "CONSOLE:" and "SYSTERM:".

APPENDIX

The following numbers correspond to device classes used in the device signature code. Make sure you contact Apple Technical Support to ensure that you have a unique device signature code.

0 -- reserved
1 -- printer
2 -- joystick or other X-Y input device
3 -- I/O serial or parallel card
4 -- modem
5 -- sound or speech device
6 -- clock
7 -- mass storage device
8 -- 80 column card

9 -- Network or bus interface
10 -- Special purpose (none of the above)

11 through 15 are reserved for future expansion

Additional Information

1. The type ahead buffer is located at $03B1 hex and is $4E hex in length. It is implemented with a read pointer (RPTR at BF18 hex) and a write pointer (WPTR at $BF19 hex). At CONSOLE: init time, these should both be set to 0. When a character is detected by CONCK, the WPTR is incremented then compared with $4E. If it is equal to $4E, it is set to $0 (this is a circular buffer). Then the WPTR is compared with RPTR and if they are equal the buffer is full. If the buffer is not full, the character is stored at $03B1+the value in WPTR.

   When removing a character from the type ahead buffer, use the following sequence. Compare the RPTR with WPTR and if they are equal, the buffer is empty and you must wait until a character is available from the keyboard. If they are not equal, increment the RPTR and compair it to $4E. If it equals $4E, set it to $0. Now get the character from location $03B1+the value in RPTR.

   If you are implementing your own type ahead, you can do it however you wish. This information is made available in case you want to check for input from another device as well as the standard system CONSOLE: and have characters from that device be put in the system type ahead buffer.

2. The example drivers in this document did not show the setting of the IORESULT in the X register. This would be done in the code specific to your driver and should allways be set to something (0 if there are no errors). If there are errors, set it as described elsewhere in this document and the Pascal Manuals.

3. For further information, see the newest edition of the Apple II Reference Manual.

4. These listings from the BIOS are included to show you how we implemented certain system drivers. You cannot rely on the locations of these to stay in the same place in the BIOS in future releases of Apple II Pascal nor can you rely on the routines themselves staying the same. They are only included as examples and to give you information that may not be documented elsewhere. This is not a complete BIOS listing so you may find references to routines or locations that are not included in this listing. The only locations that will be sure to remain the same for future releases are those mentioned in the LOCATIONS section above. We are against you poking the BIOS yourself to change or overwrite any of these routines. We did not include this information so you could poke the BIOS. If you do modify the BIOS, it is completely at your own risk! We have provided the ATTACH utility so you can add your own drivers the system without poking the BIOS and this is the way it should be done! If you have special requirements that are not solved by ATTACH, please

contact Apple Technical Support.

```
;---------------------------------------
;
; ZERO PAGE PERMANENTS
;
;---------------------------------------
FIRST .EQU 0F0 ;START ZERO PAGE USE
BAS1L .EQU FIRST ;SCREEN 1 PTR
BAS1H .EQU FIRST+1
BAS2L .EQU FIRST+2 ;SCREEN 2 PTR
BAS2H .EQU FIRST+3
CH .EQU FIRST+4 ;HORIZ CURSOR, 0..79
CV .EQU FIRST+5 ;VERT CURSOR, 0..23
TEMP1 .EQU FIRST+6
TEMP2 .EQU FIRST+7
SYSCOM .EQU FIRST+8 ;2 BYTES PTR TO SYSCOM AREA

;---------------------------------------
;
; BF00 PAGE PERMANENTS
;
;---------------------------------------
CONCKVECTOR .EQU 0BF0A ;4 BYTES
SCRMODE .EQU 0BF0E
LFFLAG .EQU 0BF0F
NLEFT .EQU 0BF11
ESCNT .EQU 0BF12
RANDL .EQU 0BF13
RANDH .EQU 0BF14
CONFLGS .EQU 0BF15
BREAK .EQU 0BF16 ;2 BYTES
RPTR .EQU 0BF18 ;1 BYTE
WPTR .EQU 0BF19 ;1 BYTE
RETL .EQU 0BF1A
RETH .EQU 0BF1B
SPCHAR .EQU 0BF1C ;00 MEANS DO ALL SPECIAL CHARACTER CHECKING
                                ;01 MEANS DON'T CHECK FOR APPLE SCREEN STUFF
                                ;02 MEANS DON'T CHECK FOR OTHER SCREEN STUFF
IBREAK .EQU 0BF1D ;INTERP STORES BREAK & SYSCOM ADR HERE FOR
ISYSCOM .EQU 0BF1F ;USER ROUTINES TO GET AT
VERSION .EQU 0BF21 ;VERSION OF SYSTEM SET TO 2 FOR APPLE 1.1
FLAVOR .EQU 0BF22 ;SEE TABLE IN INTERP BOOT
SLTTYPS .EQU 0BF27 ;BF27..0BF2E
XITLOC .EQU 0BF2F ;INTERP INITS THIS TO LOCATION OF XIT
            ;FORTRAN PROTECTION USES BF56..BF7F
            ;VENDOR BOOT DEVICES CAN USE BFC0..BFFF

;---------------------------------------
;
; MISCELANEOUS PROGRAM EQUATES
;
;---------------------------------------
BUFFER .EQU 0200 ;TEMP HSHIFT BUFFER (OVERLAPS DISK BUF)
CONBUF .EQU 03B1 ;78 CHAR TYPE-AHEAD BUF
CBUFLEN .EQU 04E ;78 DECIMAL
```

```
NCTRLS .EQU 14. ;# CTRL CHARS IN TABLE
SIGVALUE .EQU 1
BYTEPSEC .EQU 256. ;DISK INFO FOR DISKSTAT
SECPTRAK .EQU 16.
TRAKPDSK .EQU 35.
UDJVP .EQU 0E8 ;0 PAGE JUMP VECTOR POINTER LOCATIONS
DISKNUMP .EQU 0EA
JVBFOLD .EQU 0EC
JVAFOLD .EQU 0EE
HCMODE .EQU 0E1 ;THESE TWO BYTES USED FOR HIRES STUFF
HSMODE .EQU 0E0


JVECTRS .WORD UDJMPVEC
            .WORD DISKNUM
            .WORD BIOS
            .WORD BIOSAF

;--------------------------------------
;
; HARD RESET INITIALIZATION
;
;--------------------------------------
START CLD ;SET HEX MODE
            SEI ;MAKE SURE INTERRUPTS ARE OFF.

;--------------------------------------
;
; CLEAR ALL MEMORY 0 TO BFFF
; (RUN-TIME SYSTEM:0 TO TOPMEM + BF PAGE);
;
;--------------------------------------
            LDA #0
            STA ZEROL
            STA ZEROH
            TAY
            TAX
ZERLP STA (ZEROL),Y ;WRITE A BYTE OF 0
            INY ;BUMP POINTER
            BNE ZERLP ;LOOP TILL NEXT PAGE
            INC ZEROH ;BUMP MSB POINTER
            INX
            .IF RUNTIME=1
            CPX #TOPMEM ;DONE CLEARING MEM?
            BNE $1
            LDX #0BF ;CLEAR BF PAGE
            STX ZEROH
$1: CPX #0C0
            BNE ZERLP
            .ELSE
            CPX #0C0 ;DONE CLEARING BFXX?
            BNE ZERLP
            .ENDC

;----------------------
;
```

```
; CHECKSUM PROMS ON EACH SLOT
; TO FIND OUT WHO'S OUT THERE
;
; SUM TWICE TO TELL IF CARD THERE
; IF SUMS DONT MATCH THEN NO PROM IS THERE
; IF MS BYTE OF SUM=0 THEN NO PROM IS PRESENT
;
;----------------------
            LDY #0C7 ;POINT TO SLOT 7 PROM
NXTCRD STY CKPTRH ;(CKPTRL=0 FROM MEM CLEAR)
            JSR CKPAGE ;16 BIT SUM IN X,A
            STA CHECKL
            STX CHECKH ;SAVE FOR MATCH
            JSR CKPAGE ;SUM AGAIN
            CPX #0 ;WAS MSB ZERO?
            BEQ NOPROM ;YES NO PROM ON CARD
            CMP CHECKL ;LSB MATCH?
            BNE NOPROM ;NO, NO PROM ON CARD
            CPX CHECKH
            BNE NOPROM ;MSB DIDNT MATCH
            BEQ SKIPIORTS ;ALWAYS TAKEN

;---------------------------
;
; TABLE OF CN05 AND CN07 BYTES OF EACH CARD
;
;---------------------------
CN05BYTS .BYTE 003,018,038,048
CN07BYTS .BYTE 03C,038,018,048

;---------------------------
;
; NOW THAT WE KNOW A CARD IS THERE,
; EXAMINE CN05 AND CN07 BYTE TO
; DETERMINE WHICH CARD IT IS
;
; SET CARDTYPE AS FOLLOWS:
; 0=CKSUM NOT REPEATABLE OR MSB=0
; 1=CKSUM REPEATABLE,CARD NOT RECOGNIZED
; 2=DISK CARD (BYTE 07= 03C)
; 3=COM CARD (BYTE 07= 038)
; 4=SERIAL (BYTE 07= 018)
; 5=PRINTER (BYTE 07= 048)
; 6=FIRMWARE (BYTE 07= 048)
;---------------------------
SKIPIORTS LDX #5 ;4 TYPES OF CARDS
NXTYP LDY #5 ;CHECK BYTE CN05 OF CARD
            LDA (CKPTRL),Y
            CMP CN05BYTS-2,X ;MATCH TABLE?
            BNE TRYNXT ;NO, TRY NEXT IN LIST
            LDY #7
            LDA (CKPTRL),Y ;TEST CN07 BYTE
            CMP CN07BYTS-2,X ;MATCH TABLE?
            BEQ STOR ;BOTH MATCHED, CARD RECOGNIZED
TRYNXT DEX ;BUMP TO NEXT IN LIST
            CPX #2 ;TRY ALL TYPES IN LIST
```

```
            BCS NXTYP ;IF NOT IN LIST,FALL THRU WITH X=1
STOR CPX #4 ;IS IT A SERIAL CARD?
            BNE STOR1
            LDY #0B
            LDA (CKPTRL),Y
            CMP #SIGVALUE
            BNE STOR1
            LDX #6
STOR1 LDY CKPTRH
            TXA
            STA SLTTYPS-0C0,Y
NOPROM LDY CKPTRH
            DEY ;BUMP TO NEXT LOWER SLOT
            CPY #0C0 ;SLOTS 7 DOWNTO 1 DONE?
            BNE NXTCRD ;LOOP TILL 7 SLOTS DONE
                                ;LEAVE WITH Areg:=0
;-------------------------------------
;
; SET UP CONCK VECTOR FOR KEYPRESS FUNCTION
;
;-------------------------------------
            BEQ $2 ;ALWAYS BRANCHES
$1 JSR KCONCK ;HERE ARE THE 2 INSTRUCTIONS TO BE TRANSFERRED
            RTS
$2 LDY #3 ;TRANSFER 4 BYTES TO BF0A
$21 LDA $1,Y
            STA CONCKVECTOR,Y
            DEY
            BPL $21

            ;SET UP JUMP VECTOR POINTERS IN 0 PAGE
            LDY #7
$3 LDA JVECTRS,Y
            STA UDJVP,Y
            DEY
            BPL $3

;-------------------------------------
;
; SET SCREEN MODE ETC
;
;-------------------------------------
            LDA #80
            STA HOMODE
            LDA 0C051 ;SET TEXT MODE
            LDA 0C052 ;SET BOTTOM 4 GRAFIX
            LDA 0C054 ;SELECT PRIMARY PAGE
            LDA 0C057 ;SELECT HIRES GRAFIX
            LDA 0C010 ;CLEAR KEYBOARD STROBE
            JSR FORM ;ERASE SCREEN
            JSR INVERT ;PUT CURSOR ON SCREEN
            JSR DRESET ;DO ONCE ONLY DISK INIT
            LDA SLTTYPS+3 ;WHAT CARD IN SLOT 3?
            LDY #030 ;SLOT 3
            JSR GENIT ;SET BAUD IF COM OR SER THERE
            CPX #0 ;WAS AN EXTERNAL CONSOLE THERE?
```

```
            BNE STARTUP ;NO,USE APPLE SCREEN
            LDA #4
            STA SCRMODE ;SET BIT 2 FOR EXT CON
STARTUP JMP JPASCAL ;FOLD IN INTERP AND START PASCAL

;-------------------------
;
; SUB TO CHECKSUM ONE PAGE
;
CKPAGE LDA #0
            TAX ;CLEAR SUM
            TAY ;CLEAR INDEX
CKNX CLC
            ADC (CKPTRL),Y ;ADD BYTE
            BCC NOCRY
            INX ;INC HI BYTE IF CARRY
NOCRY INY ;BUMP INDEX
            BNE CKNX ;SUM 256 BYTES
            RTS ;RETURN SUM IN X,A AND Y=0


;-------------------------------------------------------------
;
; BIOS HANDLERS FOR LOGICAL AND PHYSICAL DEVICES.
;
;-------------------------------------------------------------




;--------------------------------------
;
; CONSOLE CHECK FOR CHAR AVAIL
; STATUS AND ALL REGS PRESERVED
; IF CHAR AVAIL,PUT IN CONBUF AND INC WPTR.
;
; WARNING...THIS ROUTINE ALSO CALLED FROM DISK ROUTINES
;
;--------------------------------------
CONCK PHP
            PHA
            TXA
            PHA
            TYA
            PHA
RNDINC INC RANDL ;BUMP 16 BIT RANDOM SEED
            BNE RNDOK
            INC RANDH
RNDOK LDA SLTTYPS+3 ;WHAT CARD IS IN SLOT 3?
            CMP #3 ;IS IT A COM CARD?
            BEQ COMCK ;YES,GO CHECK IT
            CMP #4 ;IS IT A SERIAL CARD?
            BEQ JDONCK ;YES,IT CANT BE TESTED
            CMP #6
            BEQ FIRMCK
TSTKBD LDA 0C000 ;TEST APPLE KEYBOARD
```

```
            BPL JDONCK ;NO CHAR AVAIL
            STA 0C010 ;CLEAR KEYBD STROBE
            AND #07F ;MASK OFF TOP BIT
            TAX ;See if checking for apple special chars is
            LDA SPCHAR ;turned off.
            ROR A
            BCS NOTFOLP2 ;Jump if so
            TXA
            CMP #11. ;CTRL-K?
            BNE NOTK
            LDA #05B ;YES,REPLACE WITH LEFT SQR BRACKETT
NOTK CMP #1 ;CTRL-A?
            BNE NTTAB
            JSR HTAB ;YES,TAB NEXT MULT 40
            LDA CONFLGS
            AND #0FE
            STA CONFLGS ;CLEAR AUTO-FOLLOW BIT
            JMP DONECK
NTTAB CMP #26. ;CTRL-Z?
            BNE NOTFOL ;NO,PUT CHAR IN BUFFER
            LDA CONFLGS
            ORA #1
            STA CONFLGS ;SET AUTO-FOLLOW BIT
            BNE DONECK ;BR ALWAYS


COMCK LDA 0C0BE ;CHAR AVAIL?
            LSR A
            BCC DONECK ;NO CHAR AVAIL
            LDA 0C0BF ;GET CHAR FROM UART
GOTCHAR AND #07F ;MASK OFF BIT 7
NOTFOL TAX
            LDA SPCHAR ;See if console special char checking is
                                ;turned off.
            ROR A
NOTFOLP2 ROR A
            BCS NFMI1 ;Jump if so
            TXA
            LDY #055
            CMP (SYSCOM),Y ;STOP CHAR?
            BNE NOTSTOP
            LDA CONFLGS
            EOR #080
            STA CONFLGS ;YES,TOGGLE STOP BIT (BIT 7)
JDONCK JMP DONECK

FIRMCK LDA #1
            LDY #030
            JSR FIRMSTATUS
            BCC DONECK
            JSR FREAD1
            JMP GOTCHAR

NOTSTOP DEY
            CMP (SYSCOM),Y
            BNE NOTBRK
```

```
            LDA CONFLGS
            AND #03F
            STA CONFLGS ;CLEAR FLUSH&STOP BITS
            .IF RUNTIME=0
             JMP TOBREAK
            .ELSE
             JMP @BREAK ;BREAK OUT
            .ENDC
NOTBRK DEY
            CMP (SYSCOM),Y ;FLUSH?
            BNE NOTFLUS
            LDA CONFLGS
            EOR #040
            STA CONFLGS ;TOGGLE FLUSH BIT (BIT 6)
            JMP DONECK

NFMI1 TXA
NOTFLUS LDX WPTR
            JSR BUMP
            CPX RPTR ;BUFFER FULL?
            BNE BUFOK
            JSR BELL
            JMP DONECK ;BEEP&IGNORE CHAR
BUFOK STX WPTR
            STA CONBUF,X ;PUT CHAR IN BUFFER
DONECK BIT CONFLGS ;IS STOP FLAG SET?
            BPL CKEXIT
            JMP RNDINC ;LOOP IF IN STOP MODE

CKEXIT PLA
            TAY
            PLA
            TAX
            PLA
            PLP
            RTS ;ELSE RESTORE STAT AND ALL REG AND RETURN
BUMP INX ;BUMP BUFFER POINTER WITH WRAP-AROUND
            CPX #CBUFLEN
            BNE BMPRTS
            LDX #0
BMPRTS RTS



;---------------------------------
;
; INITIALIZE CONSOLE:
;
;---------------------------------
CINIT PLA
            STA TEMP1 ;SAVE RETURN ADDR
            PLA
            STA TEMP2
            PLA
            STA SYSCOM ;SAVE PTR TO SYSCOM AREA
            PLA
```

```
            STA SYSCOM+1
            PLA
            STA BREAK ;SAVE BREAK ADDRESS
            PLA
            STA BREAK+1
            LDA TEMP2
            PHA ;RESTORE RETURN ADDR
            LDA TEMP1
            PHA
            LDA RPTR ;FLUSH TYPE-AHEAD BUFFER
            STA WPTR
            LDA CONFLGS
            AND #03E
            STA CONFLGS ;CLEAR STOP,FLUSH,AUTO-FOLLOW BITS
            JSR TAB3 ;NO,HORIZ SHIFT FULL LEFT
CINIT2 LDX #0 ;CLEAR IORESULT
            RTS ;AND RETURN

;--------------------------------
;
; READ FROM CONSOLE:
; KEYBOARD,COM OR SERIAL CARD IN SLOT 3
;
;--------------------------------
CREAD JSR ADJUST ;HORIZ SCROLL IF NECESSARY
            LDY #030 ;SLOT 3
            LDA SLTTYPS+3 ;WHAT TYPE OF CARD?
            CMP #4 ;IS IT A SERIAL CARD?
            BNE CREAD2 ;NO,CONTINUE
            JSR RSER ;YES, READ IT
            AND #7F ;MASK OFF TOP BIT
            RTS
CREAD2 JSR CONCK ;TEST CHAR
            LDX RPTR
            CPX WPTR
            BEQ CREAD ;LOOP TILL SOMETHING IN BUFFER
            JSR BUMP
            STX RPTR ;BUMP READ POINTER
            LDA CONBUF,X ;GET CHAR FROM BUFFER
            LDX #0 ;CLEAR IORESULT
            RTS ;AND RETURN TO PASCAL

;--------------------------------
;
; INITIALIZE PRINTER:
; PRINTER IS ALWAYS IN SLOT 1
; IT MAY BE A PRINTER,COM,OR SERIAL CARD
;
;--------------------------------
PINIT LDY #010 ;SLOT 1 ; 010
            LDA SLTTYPS+1 ;WHAT CARD IN SLOT 1?
            CMP #5 ;PRINTER CARD?
            BEQ CLRIO1 ;YES,NO INIT NEEDED
GENIT CMP #4 ;SERIAL CARD?
            BEQ ISER ;YES, INIT SER CARD
            CMP #3 ;COM CARD?
```

```
            BEQ ICOM ;YES,INIT COM CARD
            CMP #6
            BEQ FIRMINIT
            LDX #9 ;NONE OF ABOVE,OFFLINE
            RTS

FIRMINIT PHA
            JSR SER1
            LDY #0D
FVEC1 LDA (TEMP1),Y
            STA TEMP1
            LDY 6F8
            PLA
            JMP @TEMP1

;---------------------------------
;
; INITIALIZE REMOTE:
; REMOTE IS ALWAYS IN SLOT 2
; IT MAY BE A COM OR SERIAL CARD
;
;---------------------------------
RINIT LDA SLTTYPS+2 ;WHAT CARD IN SLOT 2?
            LDY #020
            BNE GENIT ;BR ALWAYS TAKEN

;---------------------------------
;
; INIT COM CARD, Y=ONO
;
;---------------------------------
ICOM LDA #3 ;MASTER INIT
            STA 0C08E,Y ;TO STATUS
            LDA #21.
            STA 0C08E,Y ;SET BAUD ETC
CLRIO1 LDX #0 ;CLEAR IORESULT
            RTS ;AND RETURN

;---------------------------------
;
; INIT SERIAL CARD, Y=ONO
;
;---------------------------------
ISER JSR SER1 ;ASSORTED GARBAGE
            JSR 0C800 ;SET UP SLOT DEPENDENTS
CLRIO3 LDX #0 ;CLEAR IORESULT
            RTS ;AND RETURN

;---------------------------------
;
; ASSORTED SERIAL CARD SET-UP
;
;---------------------------------
SER1 STY 06F8 ;STORE NO
            TYA
            LSR A
```

```
            LSR A
            LSR A
            LSR A
            ORA #0C0
            TAX ;MAKE OCN IN X
            LDA #0
            STA TEMP1
            STX TEMP2 ;SET UP INDIRECT ADDRESS
            LDA 0CFFF ;TURN OFF ALL C8 ROMS
            LDA (TEMP1),Y ;SELECT C8 BANK
            RTS

;--------------------------------
;
; WRITE TO CONSOLE:
; VIDEO SCREEN,COM OR SER CARD IN SLOT 3
;
;--------------------------------
CWRITE JSR CONCK ;CONSOLE CHAR AVAIL?
            BIT CONFLGS ;IS FLUSH FLAG SET?
            BVS CLRIO ;YES,DISCARD CHAR & RETURN
            TAX ;SAVE CHAR IN X
            LDY #030 ;SLOT 3;010
            LDA SLTTYPS+3 ;WHAT KIND OF CARD?
            CMP #3 ;COM CARD?
            BEQ WCOM ;YES WRITE TO COM CARD SLOT 3
            CMP #4 ;SERIAL CARD?
            BEQ WSER ;YES,WRITE TO SER CARD SLOT 3
            CMP #6
            BEQ WFIRM
            TXA ;ELSE RESTORE CHAR & SEND TO SCREEN
VIDOUT STA TEMP1 ;SAVE CHAR FOR LATER
            JSR INVERT ;REMOVE CURSOR
            LDY CH
            JSR VOUT2 ;DO THE BUSINESS
            JSR INVERT ;RESTORE THE CURSOR
CLRIO LDX #0 ;CLR IORESULT
            RTS ;RETURN FROM VIDOUT

WFIRM TXA
            PHA
            LDA #0
            JSR IOWAIT
            JSR SER1
            LDY #0F
            JMP FVEC1

;--------------------------------
;
; WRITE TO SERIAL CARD, Y=ONO,CHAR IN X
;
;--------------------------------
WSER JSR CONCK ;CONSOLE CHAR?
            TXA
            PHA ;SAVE CHAR ON STACK
            JSR SER1 ;ASSORTED GARBAGE
```

```
            PLA
            STA 05B8,X ;SET UP DATA BYTE
            JSR 0C9AA ;SEND IT (SHOUT)
            LDX #0
            RTS

;---------------------------------
;
; WRITE TO REMOTE:, CHAR IN A
;
;---------------------------------
RWRITE TAX ;SAVE CHAR
            LDA SLTTYPS+2 ;WHAT CARD IN SLOT 2?
            LDY #020
            BNE GENW2 ;BR ALWAYS TAKEN

;---------------------------------
;
; WRITE TO PRINTER CARD SLOT1, CHAR IN X
;
;---------------------------------
WPRN JSR CONCK ;CONSOLE CHAR AVAIL?
            LDA 0C1C1 ;TEST PRINTER READY
            BMI WPRN ;LOOP TILL READY
            STX 0C090 ;SEND CHAR
CLRIO2 LDX #0
            RTS

;---------------------------------
;
; WRITE TO COM CARD, Y=ON0,CHAR IN X
;
;---------------------------------
WCOM JSR CONCK ;CONSOLE CHAR?
            LDA 0C08E,Y ;TEST UART STATUS
            AND #2 ;READY?
            BEQ WCOM ;NO,WAIT TILL READY
            TXA
            STA 0C08F,Y ;SEND CHAR
            LDX #0
            RTS

;---------------------------------
;
; WRITE TO PRINTER:, CHAR IN A
;
;---------------------------------
PWRITE TAX ;SAVE CHAR IN X
            LDA LFFLAG ;TEST LINE-FEED FLAG
            BPL LFPASS ;PASS IF BIT7=0
            CPX #10. ;IS IT A LINE-FEED?
            BEQ CLRIO ;YES,IGNORE
LFPASS LDY #010 ;SLOT 1
            LDA SLTTYPS+1 ;WHAT KIND OF CARD?
GENW CMP #5 ;PRINTER CARD?
            BEQ WPRN ;YES WRITE TO PRINTER CARD
```

```
GENW2 CMPL#4 ;SERIAL CARD?
              BEQ WSER ;YES WRITE TO SER CARD
              CMP #3 ;COM CARD?
              BEQ WCOM ;YES WRITE TO COM CARD
              CMP #6
              BEQ WFIRM
OFFLINE LDX #9
              RTS

;---------------------------------
;
; READ FROM REMOTE:
;
;---------------------------------
RREAD LDA SLTTYPS+2 ;WHAT CARD IN SLOT 2?
              LDY #020
GENR CMP #4 ;SERIAL CARD?
              BEQ RSER ;GET FROM SER CARD
              CMP #3 ;COM CARD?
              BEQ RCOM ;GET FROM COM CARD
              CMP #6
              BEQ RFIRM
              BNE OFFLINE ;CARD NOT RECOG

;---------------------------------
;
; READ FROM COM CARD, Y=NO
;
;---------------------------------
RCOM JSR CONCK ;CHECK FOR CONSOLE CHAR
              LDA 0C08E,Y ;TEST UART STATUS
              LSR A ;TEST BIT 0
              BCC RCOM ; WAIT FOR CHAR
              LDA 0C08F,Y ;GET CHAR
              LDX #0
              RTS

RFIRM LDA #1
              JSR IOWAIT
FREAD1 JSR SER1
              PHA
              LDY #0E
              JMP FVEC1

;---------------------------------
;
; READ FROM SERIAL CARD, Y=ONO
;
;---------------------------------
RSER JSR CONCK ;CONSOLE CHAR AVAIL?
              JSR SER1 ;ASSORTED GARBAGE
              JSR 0C84D ;GET A BYTE (SHIFTIN)
              LDA 05B8,X ;GET BYTE 0678+SLOT
              LDX #0
              RTS
```

```
FIRMSTATUS PHA
           JSR SER1
           LDY #10
           JMP FVEC1

IOWAIT JSR CONCK
           PHA
           JSR FIRMSTATUS
           PLA
           BCC IOWAIT
           RTS
```

notes